Adversarial Search Monte-Carlo-Search Trees

Seach, Trees, etc

Monte-Carlo

Idea

When a problem is too hard to solve exactly, simulate it randomly thousands of times and observe the outcomes.

Classic example: estimating π:

- Imagine a square with a circle inside it. You throw darts randomly at the square. Some land inside the circle, some outside. The ratio of darts inside vs total darts approximates π/4.

Darts inside circle / Total darts ≈ π/4

Example Tic-Tac-Toe, mid game

- It's your turn (X). The board looks like this:

```
X | _ | O
---------
_ | X | _
---------
_ | _ | O
```

```
[0,0] | [0,1] | [0,2]
---------------------
[1,0] | [1,1] | [1,2]
---------------------
[2,0] | [2,1] | [2,2]
```

- **100 sims** = 100 complete games played out from that position, with both players making random moves until someone wins or it's a draw.

Sim 1: You play [1,2] → random → random → random → X wins ✓

- Cell [0,1] → 100 sims → 31 wins → 31%
- Cell [1,0] → 100 sims → 28 wins → 28%
- Cell [1,2] → 100 sims → 67 wins → 67% ← pick this
- Cell [2,0] → 100 sims → 44 wins → 44%
- Cell [2,1] → 100 sims → 19 wins → 19%

Monte Carlo Search Trees (MCST)

An alternative to alpha–beta search for games like Go where the branching factor is enormous and good evaluation functions are hard to design.

The core idea: MCTS estimates a state's value by *simulating complete games* from that state and seeing who wins. If you play 100 random games from a position and black wins 60 of them, you estimate black's win probability at 60%.

Playout policy: Purely random play would answer "what's the best move if both players play randomly?" which is usually not the same as "what's the best move if both players play well?" So MCTS uses a *playout policy* that biases moves toward good ones, often learned through neural networks and self-play.

MCST steps

- SELECTION: Starting at the root of the search tree, we choose a move (guided by the selection policy),
- EXPANSION: We grow the search tree by generating a new child of the selected node.
- SIMULATION: We perform a playout from the newly generated child node, choosing moves for both players according to the playout policy. These moves are *not* recorded in the search tree, just the final result.
- BACK-PROPAGATION: We now use the result of the simulation to update all the search tree nodes going up to the root.

Selection policy

The UCB1 formula is what makes the selection step smart. For each node *n*, it computes:

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

UCB1 formula

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

Exploitation term **Exploration term**

U(n) / N(n) — Exploitation term

U(n) = total wins through node n.

N(n) = total playouts through node n.

This is just the **win rate**. It favors nodes that have performed well so far. A node with 60 wins out of 79 playouts scores 0.76 — that looks good, keep going there.

C × √(log N(parent) / N(n)) — Exploration term

N(parent) = total playouts through the parent node.

N(n) = total playouts through this node.

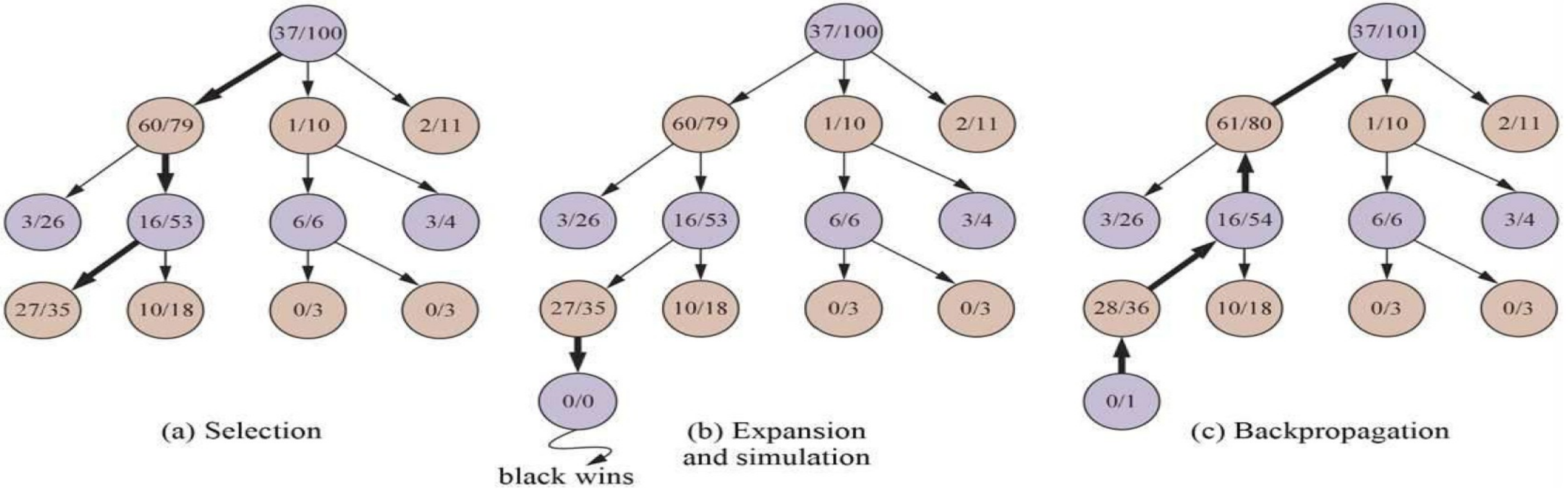
C = a constant (theoretically √2 ≈ 1.41) that controls the balance.

The key insight: **log(parent visits) / node visits**. If the parent has been visited 100 times but this child only 11 times, the ratio is high — we haven't explored this child enough. As N(n) grows, this term shrinks toward zero, and exploitation dominates.

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

Exploitation term **Exploration term**

Find UCB1 for the three children of the root

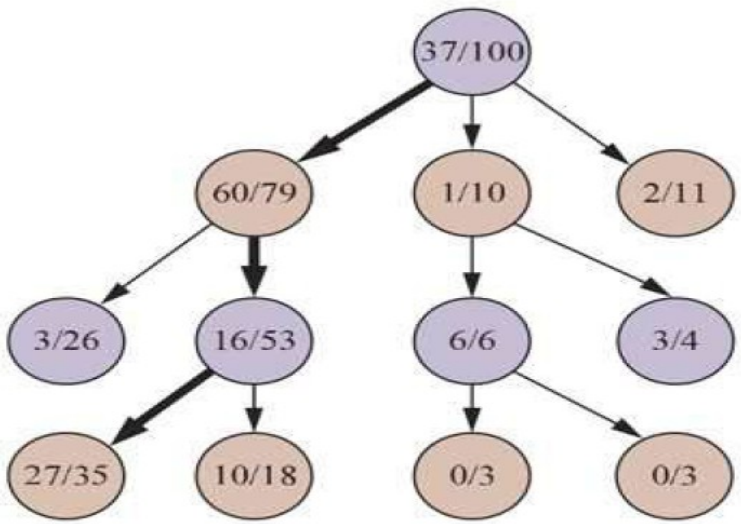


$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

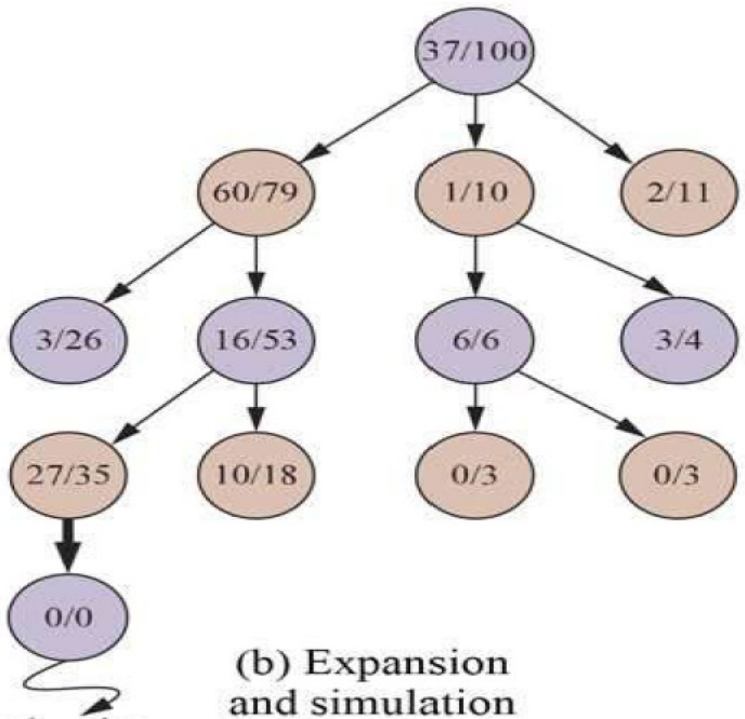
Exploitation term **Exploration term**

60/79	1/10	2/11
win rate: 0.759	win rate: 0.100	win rate: 0.182
explore: +0.338	explore: +0.950	explore: +0.906
1.098	1.050	1.088
← selected		

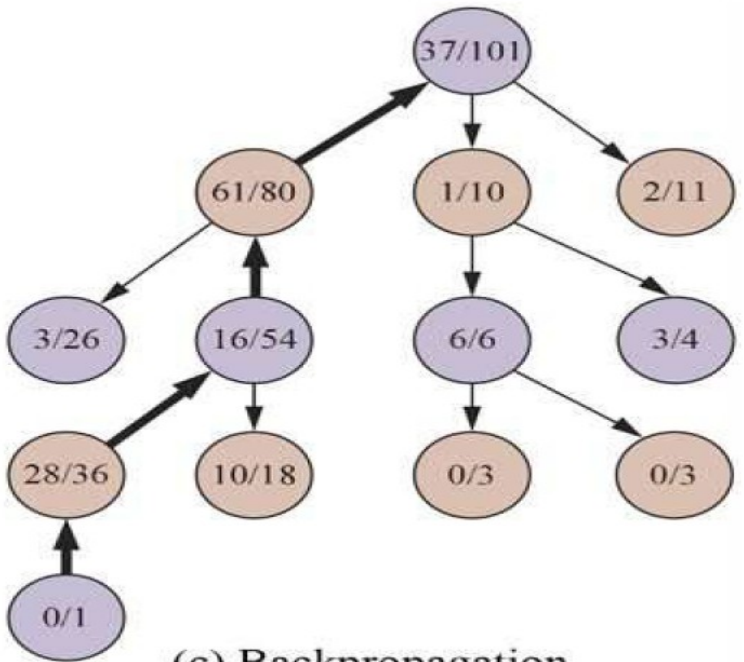
● Exploitation (win rate) ● Exploration (uncertainty bonus)



(a) Selection

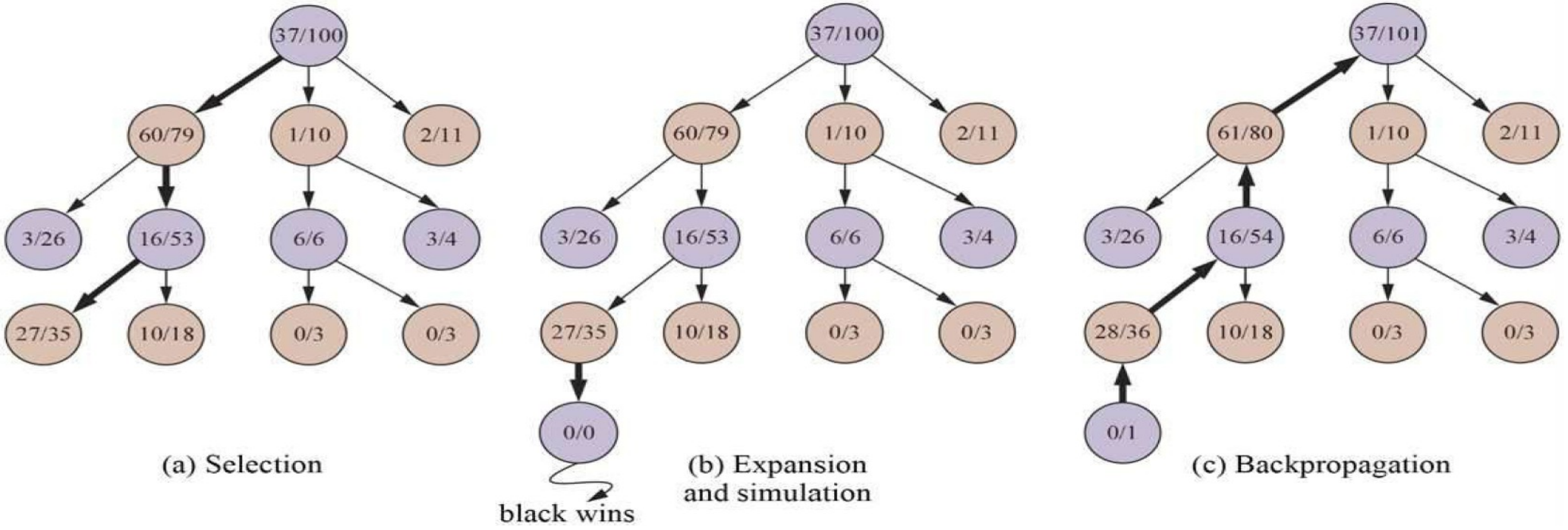


(b) Expansion and simulation



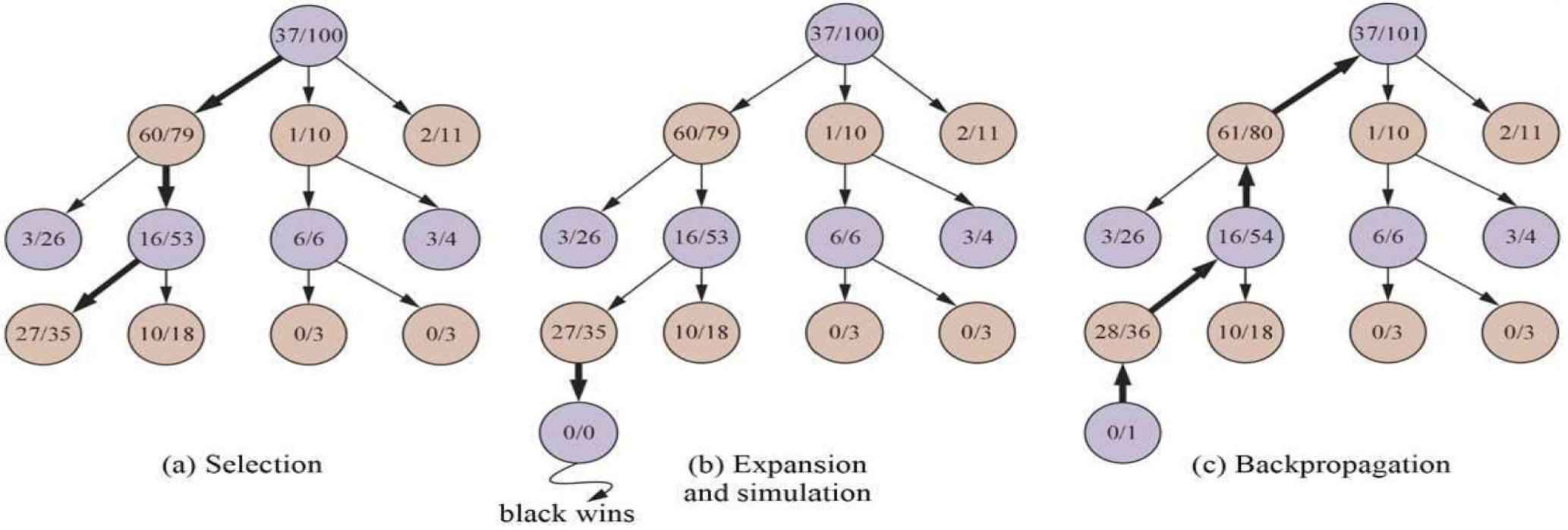
(c) Backpropagation

Selection policy: Balances (exploitation and exploration)



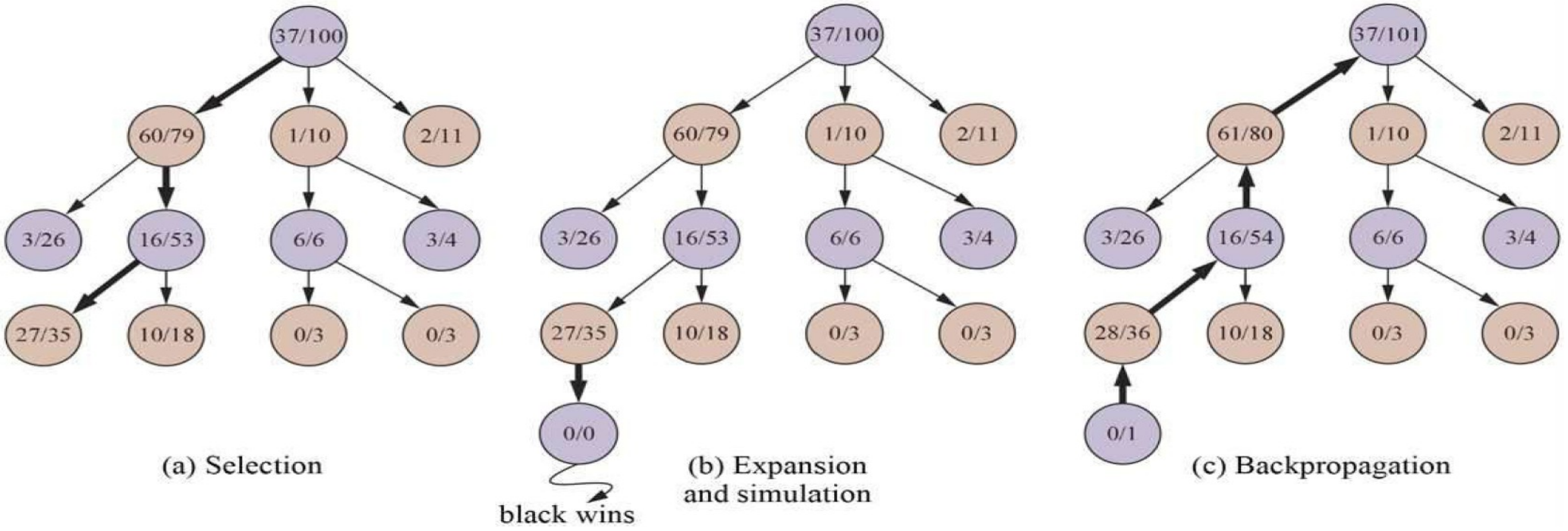
The root represents a state where white has just moved, and white has won 37 out of the 100 playouts done so far. The thick arrow shows the selection of a move by black that leads to a node where black has won 60/79 playouts. This is the best win percentage among the three moves, so selecting it is an example of exploitation.

Expansion: Balances (exploitation and exploration)



The selected leaf node (27/35) is expanded by generating one new child node. This new node represents an unexplored game state and starts with a record of 0/0, no wins, and no playouts yet.

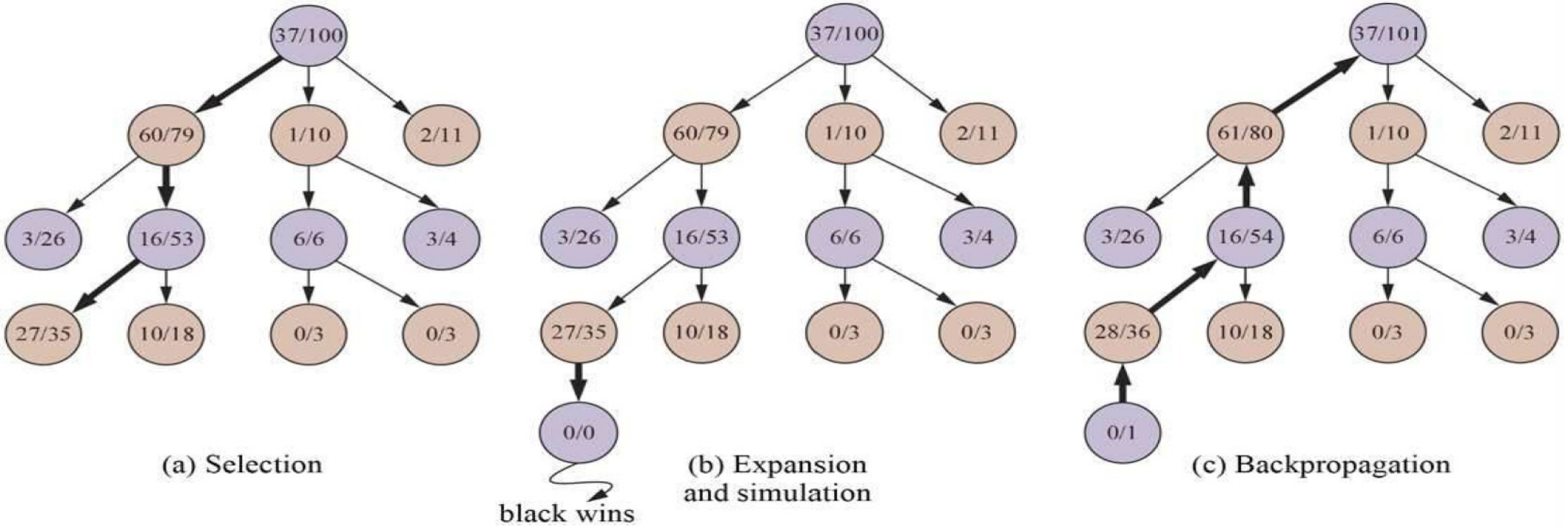
Simulation: plays out using the playout policy



From the new 0/0 node, MCTS plays out a complete game using the playout policy, choosing moves for both sides until someone wins or the game ends. These simulated moves are not added to the search tree. In this example, the playout results in a win for black.

Back-propagation

$$UCB1(n) = \underbrace{\frac{U(n)}{N(n)}}_{\text{Exploitation term}} + \underbrace{C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}}_{\text{Exploration term}}$$



The simulation result (black wins) is propagated back up through every node on the path to the root. Black's nodes get +1 win and +1 playout. White's nodes get +0 wins but still +1 playout. This updates the win rates that the selection policy uses in the next iteration

The Algorithm

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
    tree ← NODE(state)
    while IS-TIME-REMAINING() do
        leaf ← SELECT(tree)
        child ← EXPAND(leaf)
        result ← SIMULATE(child)
        BACK-PROPAGATE(result, child)
    return the move in ACTIONS(state) whose node has highest number of playouts
```

//Create the root node from the current game state. Starts with 0 wins and 0 playouts.

//Keep iterating as long as we have time. Each pass through the loop does one SELECT → EXPAND → SIMULATE → BACK-PROPAGATE cycle. More iterations = better move

//Starting at the root, walk down the tree by picking the child with the highest UCB1 score at each level. This continues until we reach a leaf

//Generate one new child node from the selected leaf. This child represents a game state one move beyond the leaf; a state we haven't simulated from yet. It's initialized as 0/0 (no wins, no playouts). The function returns this new node, so we can simulate from it next.

//From the new child node, play out a complete game to the end. Both sides make moves. until a terminal state is reached.

//Walk back up from the child node to the root, updating every node along the path

The Monte Carlo tree search algorithm. A game tree, *tree*, is initialized, and then we repeat a cycle of SELECT / EXPAND / SIMULATE / BACK-PROPAGATE until we run out of time, and return the move that led to the node with the highest number of playouts.

Monte-Carlo

Works well

New/unknown games

High branching factor

No heuristic needed

More time = better answer

Struggles

Decisive single moves

Subtle tactical sequences

When random play is misleading

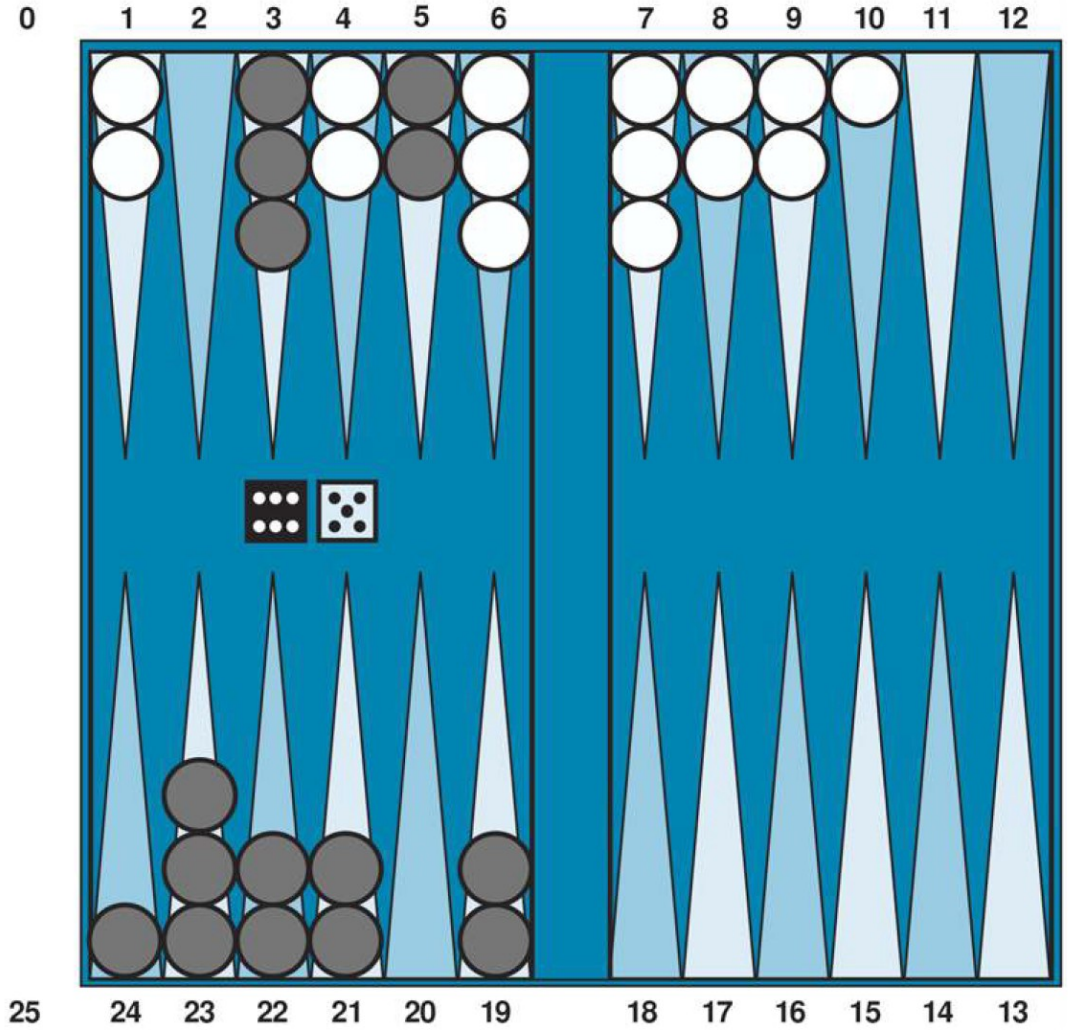
Wastes sims on bad moves equally

wastes sims on bad moves equally, this is exactly the problem Monte-Carlo Tree Search (MCST) fixes.

Stochastic Games

- **The core problem:** All the games we've seen so far (chess, Go, tic-tac-toe) are deterministic; the outcome depends only on the players' choices. Stochastic games add a **random element**, like dice. Backgammon is an example: you roll two dice before each move, and the roll determines which moves are legal.

Example: Backgammon



Black has rolled **6 − 5** and must choose among four legal moves: (5 − 11, 5 − 10) , (5 − 11, 19 − 24) , (5 − 10, 10 − 16), and (5 − 11, 11 − 16), where the notation means move one piece from position 5 to 11, and then move a piece from 11 to 16.

At this point, Black knows what moves can be made, but does not know what White is going
to roll and thus does not know what White's legal moves will be. That means Black cannot
construct a standard game tree of the sort we saw in chess and tic-tac-toe.

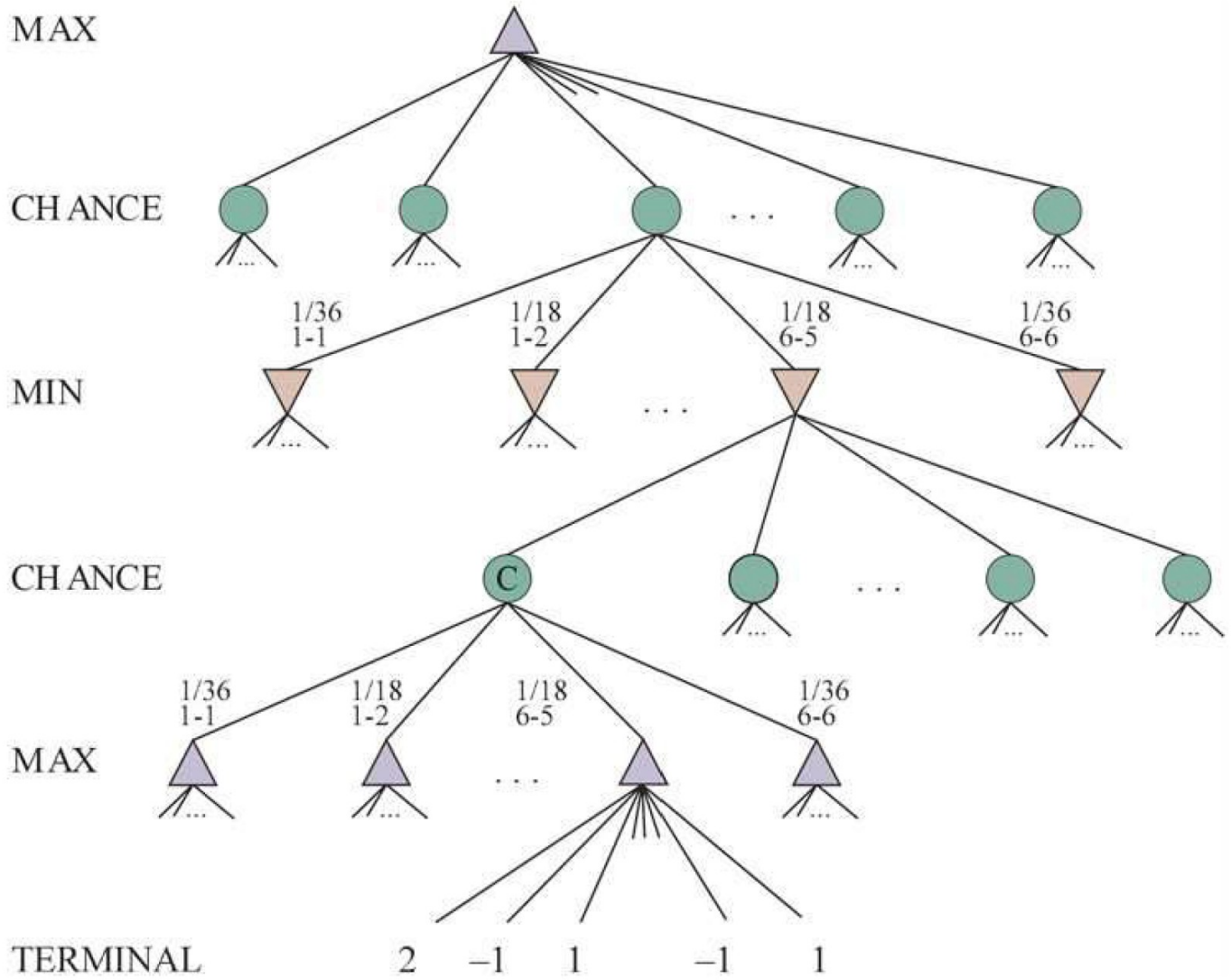
Expected minimax

- **Chance nodes:** The game tree now has three types of nodes instead of two. MAX nodes (your turn), MIN nodes (opponent's turn), and chance nodes (circles in the tree) representing dice rolls.

- **Expectiminimax:** This replaces minimax. It works identically for MAX nodes (pick the max) and MIN nodes (pick the min), but for chance nodes, it computes the expected value, the weighted average over all possible dice rolls.

- Thus, the **expectiminimax value** for games with chance nodes, is a generalization of the minimax value for deterministic games.

Expectiminimax value tree



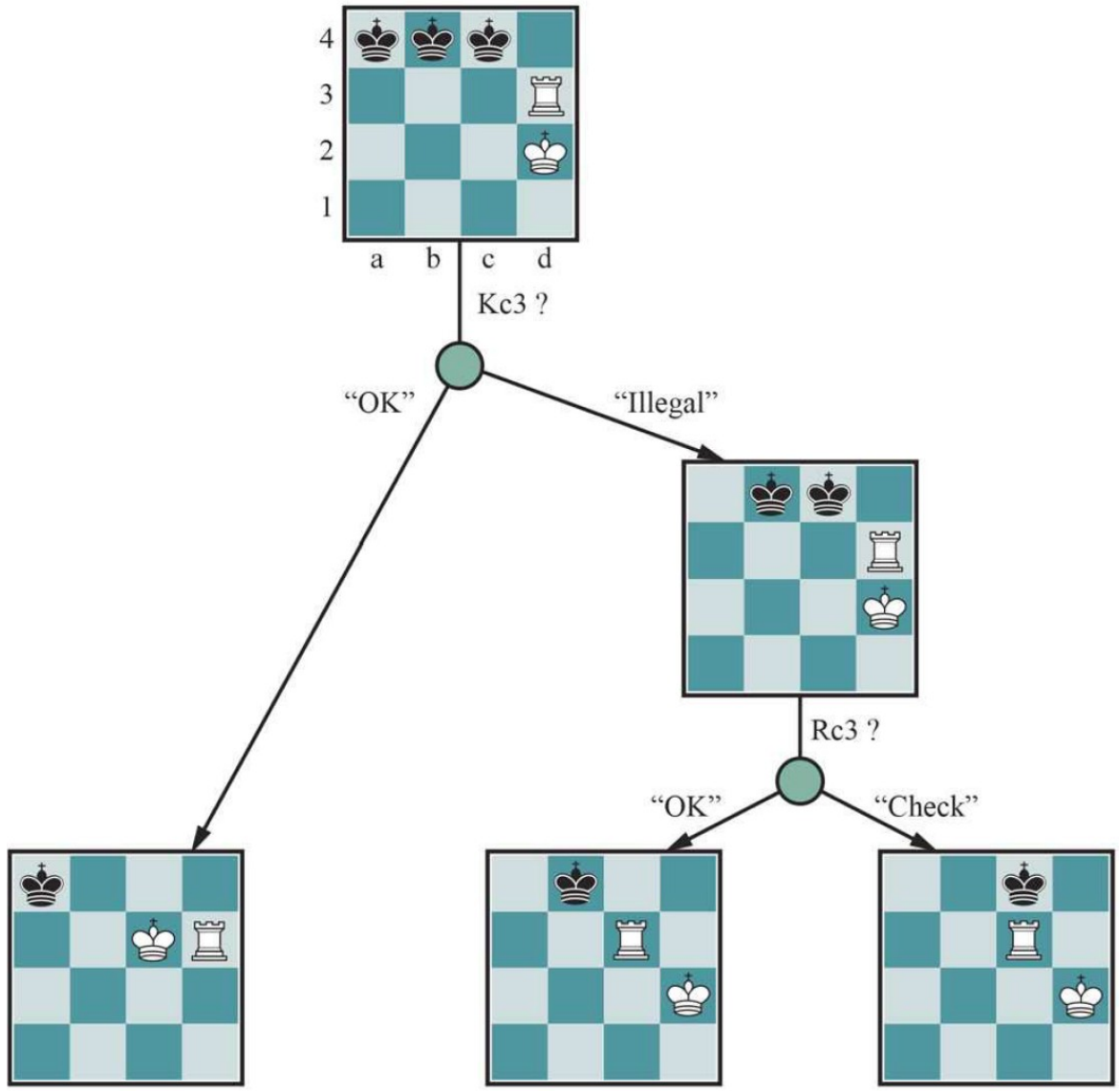
Schematic game tree for a backgammon position.

Partially Observable Games

In *deterministic* partially observable games, uncertainty about the state of the board arises entirely from lack of access to the choices made by the opponent. This class includes:

- children's games such as Battleship (where each player's ships are placed in locations hidden from the opponent)
- Stratego (where piece locations are known but piece types are hidden).
- **Kriegspiel**, a partially observable variant of chess, in which pieces are completely invisible to the opponent

Kriegspiel



Limitations of Game Search Algorithms

- Limitation 1: The evaluation function can be wrong: Alpha-beta pruning relies on a scoring function that estimates how good a position is. Think of it like a real estate appraiser estimating the value of a house. If the appraisal is off by even a small amount, you might pick the wrong house. In games, the algorithm trusts those scores completely. If the scoring function says position A is worth 100 and position B is worth 99, the algorithm picks A every time. The algorithm has no way to account for this uncertainty.

Limitations of Game Search Algorithms

- Limitation 2: It wastes time on obvious decisions. Imagine you are at a restaurant and the menu has only one dish. You do not need to spend twenty minutes weighing the options; you just order it.
- Both algorithms lack this common sense. Even when there is only one legal move, or one move is clearly the best, they still spend time computing the exact value

Limitation 3: It thinks one move at a time, never in terms of plans.

- A human chess player might think, "I want to trap the opponent's queen in the next five moves," and then work backward to figure out how. Both minimax and MCTS never think this way. They evaluate each move individually, without forming a higher-level strategy.

More limitation

- Limitation 4: Early programs depended on human experts to tell them what mattered. If the human got it wrong, the program played badly.

- Limitation 5: The horizon effect: For example, if the algorithm searches 10 moves ahead, the opponent can sacrifice a piece to create a distraction that pushes the real threat to move 11. The algorithm sees a calm position at move 10 and thinks everything is fine, when in reality, it is about to lose.

- Limitation 6: MCTS can miss rare but critical moves.

Finally, they were designed for simple, clean games.

Mini project 3

- Check the mini project 3. Create agents to play Tic-Tac-Toe using MinMax trees with Alpha-beta pruning, and Monte-Carlo Search Tree (MCST)